Optimize Unsynchronized Garbage Collection in an SSD Array

Da Zheng, Randal Burns
Department of Computer Science
Johns Hopkins University

Alexander S. Szalay
Department of Physics and Astronomy
Johns Hopkins University

Abstract

Solid state disks (SSDs) have advanced to outperform traditional hard drives significantly in both random reads and writes. However, heavy random writes trigger frequent garbage collection and decrease the performance of SSDs. In an SSD array, garbage collection of individual SSDs is not synchronized, leading to underutilization of some of the SSDs.

We propose a software solution to tackle the unsynchronized garbage collection in an SSD array installed in a host bus adaptor (HBA), where individual SSDs are exposed to an operating system. We maintain a long I/O queue for each SSD and flush dirty pages intelligently to fill the long I/O queues so that we hide the performance imbalance among SSDs even when there are few parallel application writes. We further define a policy of selecting dirty pages to flush and a policy of taking out stale flush requests to reduce the amount of data written to SSDs. We evaluate our solution in a real system. Experiments show that our solution fully utilizes all SSDs in an array under random write-heavy workloads. It improves I/O throughput by up to 62% under random workloads of mixed reads and writes when SSDs are under active garbage collection. It causes little extra data writeback and increases the cache hit rate.

1 Introduction

Solid state disks (SSDs) achieve great success due to significant performance improvement over traditional hard drives in random I/O. However, due to hardware limitation, SSDs require an expensive erase operation before writing data to used blocks. The granularity of the erase operation is usually multiple pages. To counter the cost of erase, most SSDs use a log structure to organize data and have the Flash Translation Layer (FTL) to map data to physical locations on an SSD. Thus, SSDs require garbage collection to clean space after substantial data write. Heavy random writes trigger frequent garbage collection and slow down SSDs.

Much effort has been made to reduce overhead of garbage collection in SSDs [3, 8, 6, 1, 5] and SSD vendors also add much intelligence to their firmware. They all achieve a certain degree of success, but the overhead of garbage collection can never be eliminated completely.

In an SSD array, unsynchronized garbage collection in individual SSDs leads to performance degradation. Due to the unsynchronized garbage collection, SSDs of the same model have different throughput at any particular moment. Both hardware RAID controllers and the software RAID in the Linux kernel only allow a limited number of pending I/O requests. As a result, even though the I/O queue in the RAID controller or the software RAID is filled with requests, some SSDs may still starve for requests. Such performance imbalance among SSDs leads to underutilization of some of the SSDs.

A possible solution is to synchronize garbage collection among SSDs. Such a solution requires extra hardware added to SSDs and RAID controllers, as suggested by Kim et al. [4]. Therefore, it requires coordination of SSD vendors and RAID controller vendors. It can hardly become reality and benefit end users in a short future.

We propose a software solution to tackle the unsynchronized garbage collection in an SSD array and implement our solution in the set-associative filesystem (SAFS) [12], designed to provide maximal performance of an SSD array. It is a general solution and does not rely on any specific SSD characteristics. Instead of using RAID controllers, we attach SSDs to host bus adapters (HBA) and expose individual SSDs to an operating system. We maintain a short high-priority I/O queue for application requests and a long low-priority I/O queue for flush requests in the main memory for each SSD. The short high-priority I/O queues keep the latency of application I/O requests low, while the long low-priority I/O queues hide the performance imbalance among SSDs

caused by garbage collection. We utilize the page cache in SAFS to absorb application writes and design a flushing scheme to write back dirty pages intelligently. We further define a policy of selecting dirty pages to flush and a policy of taking out stale flush requests to reduce the amount of data written to SSDs.

Experiments show promising results. The design fully utilizes all SSDs in an array and improves the performance of SAFS under random write-heavy workloads. It increases the I/O throughput of SAFS by up to 64% under mixed read/write workloads. The design increases the cache hit rate and flushes insignificant amount of extra data to SSDs.

2 Related Work

There is enormous amount of work on reducing overhead of garbage collection on a single SSD. For instance, SFS [8] is a file system specifically designed for SSDs to reduce overhead of garbage collection. It groups data blocks with similar update likelihood into the same segments to reduce the amount of data copied in garbage collection. BPLRU [3] is a buffer management scheme for the firmware inside SSDs. It uses a block-level LRU to manage the write buffer, and a page padding technique when flushing victim blocks. In-page logging (IPL) [6] is a buffer management scheme designed for DBMS. It reserves some space in each erase block of an SSD to log small writes to the block and reconstructs data for reads. Our solution works on multiple SSDs and treats each SSD as a black box, so it can be well integrated with these techniques.

Kim et al. [4] suggested to build an SSD-aware RAID controller and SSD devices capable of global garbage collection to synchronize garbage collection in an SSD array. Their solution requires the advance of both SSD devices and RAID controllers and they evaluation their solution with simulation. In contrast, we provide a software solution for commodity hardware and have an implementation in a real system for evaluation. It benefits users immediately.

3 Design

Our solution extends our previous work on SAFS [12], a user-space filesystem designed to achieve maximal performance from an SSD array in a NUMA machine, to tackle unsynchronized garbage collection in an SSD array. The root of inefficiency in an SSD RAID under garbage collection is the limited size of the I/O queue of the RAID. The SSDs under active garbage collection cannot keep up with other SSDs in the RAID and the overall performance of the SSD RAID is limited by the slowest SSD. Therefore, our solution increases I/O queues in SAFS and deploys a dirty page flusher to achieve maximal performance from an SSD array with a small number of parallel I/Os. Currently, we implement our solution in the user space.

3.1 Architecture

The architecture of SAFS in Figure 1 has five components: the file abstraction interface, the page cache, the data mapping layer, I/O queues and I/O threads. SAFS exposes a file abstraction interface to applications to receive I/O requests and notify the applications of the completion of requests. Currently, it supports an asynchronous I/O interface. SAFS is equipped with a lightweight, scalable page cache called SA-cache [11], where pages are grouped into many small page sets. As shown by Zheng et al. [11], Linux page cache has very high locking overhead in a large parallel machine when the page turnover rate is high, due to the global locks on the page cache. By grouping pages into many small page sets, SA-cache eliminates the locking overhead. Beneath the page cache is the data mapping layer, which splits and dispatches I/O requests to SSDs. SSDs are connected to the machine via host bus adapters (HBA), thus individual SSDs are exposed to the operating system. Each SSD has a native filesystem to manage the data stored on the SSD. It also has a dedicated I/O thread and originally has only one dedicated I/O queue to buffer I/O requests. Concurrent access to SSDs causes significant lock contention in the block subsystem of an operating system. The dedicated I/O threads reduce the lock contention in the operating system when issuing I/O requests to SSDs.

To tackle unsynchronized garbage collection in an SSD array, we modify the I/O queues associated with SSDs and add a dirty page flusher to the page cache of SAFS, shown as the shaded components in Figure 1. We split the original I/O queue of an SSD into two queues: a short high-priority queue and a long low-priority queue. The dirty page flusher pre-cleans dirty pages in the page cache and issues parallel write requests to SSDs.

3.2 I/O queues and prioritized I/O requests

SAFS [12] maintains an I/O queue for each SSD in the main memory, and these I/O queues can be made substantially large to hide performance disparity among SSDs. When some SSDs stall due to active garbage collection, application requests can still be dispatched to any I/O queue. Therefore, applications are not blocked by the garbage collection in some SSDs.

However, simply increasing the length of I/O queues cannot completely solve the problem. Only applications capable of issuing many parallel I/O requests can bene-

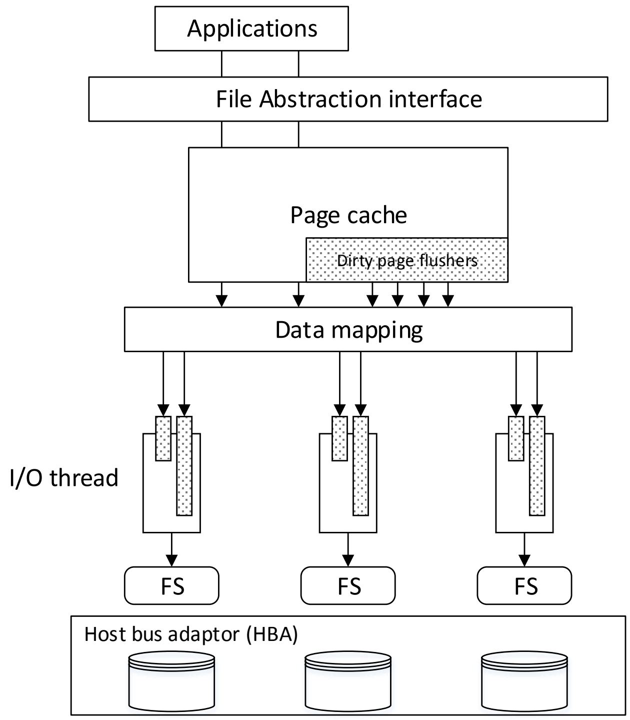

Figure 1: The architecture of SAFS. The shaded components of SAFS are modified to tackle unsynchronized garbage collection.

fit from the large I/O queues. Therefore, we flush dirty pages in the page cache to fill the long I/O queues. In a mixed read/write workload, the I/O queues are filled with application read requests and flush requests. It leads to long latency in application reads.

The solution is to split each I/O queue into two queues to provide different service quality for different types of I/O requests. One contains high-priority interactive I/O requests (application reads) and the other contains low-priority background I/O requests (flush requests). Only when there are no high-priority requests, the I/O threads issue the low-priority requests to SSDs. Hence, application reads get much shorter service time. It is essential to reduce the service time for application reads in the case of read-update-write. Any unaligned write requires read-update-write. Reducing the service time of reads allows applications to perform read-update-writes at a higher rate.

To further reduce the service time of application requests, we always reserve some I/O slots on each SSD for application requests even if there are no application requests at a moment. Such a decision is made based on the fact that SSDs can run at decent performance even if they do not receive the maximal number of parallel I/O requests required by the SSDs. When application requests are added to the high-priority queue, they are issued to SSDs immediately. An SSD typically requires 32 parallel I/O requests to achieve maximal performance and we empirically reserve seven I/O slots for high-priority I/O requests.

3.3 A dirty page flusher

The task of the dirty page flusher is to issue many flush requests to fill the I/O queues while keeping the amount of data written back to SSDs small. Filling the I/O queues with flush requests potentially leads to writing much more data to SSDs than necessary. It is essentially important to reduce data writeback because it helps increase the application-perceived I/O throughput and reduce SSD wear-out.

The set-associative cache in SAFS composes of many small page sets and the dirty page flusher is triggered to write back dirty pages in page sets where the number of dirty pages exceeds a threshold. We empirically set the size of a page set to 12 [12] and set the threshold to 6. The flusher writes back only a small number of (one or two) dirty pages from a page set each time. A page set that contains more dirty pages for writing back will be placed in a FIFO queue. Once some flush requests complete, the flusher checks the page sets in the queue in a round-robin manner and issue more flush requests until no pages can be flushed in the page sets. The algorithm gives each page set a chance to flush dirty pages but is biased in favor of the page sets that get more writes.

The dirty page flusher together with the page cache reduces the average latency of application writes, thus dramatically reducing the number of parallel application writes required to achieve good performance. When application writes hit page cache, they return immediately if the required pages exist in the cache. In the case of cache misses, writes can also return immediately if the evicted pages are clean. Application writes may trigger page writes to SSDs if the victim page is dirty, and they have to wait until the page writes to SSDs complete. With the help of the dirty page flusher, the page cache maintains a certain number of clean pages. Therefore, majority of writes are absorbed by the page cache and return immediately. To further reduce the latency of application writes, we tweak page eviction policies in SAFS to favor evicting clean pages, similar to clean-first LRU [9].

Clean-first page eviction policies may reduce the cache hit rate, as they ignore dirty pages when clean pages exist, and the dirty page flusher alleviates the problem. The dirty page flusher writes back dirty pages that are most likely to be evicted based on the page eviction policy. Once the data of a dirty page is written back to an

SSD, the page is likely to be evicted. As a result, we essentially run the page eviction policy on clean pages and dirty pages separately. More sophisticated cache management policies such as [7, 2] may be used to better balance read and write performance.

The flush requests in a long low-priority I/O queue are subject to long latency and may be discarded. Given the length of a low-priority queue, a flush request may take a long time to reach the head of the queue. When it does, it may have become stale because the page in the request may have been written back to SSDs or is no longer urgent to be flushed based on the page eviction policy. It is computationally expensive to sort all flush requests in the I/O queue to find the most urgent ones to flush. Instead, we simply discard all stale flush requests, which gives more urgent flush requests a better chance to be written to SSDs. Once discarding stale flush requests, an I/O thread will notify the page cache and ask for more flush requests. The scheme of discarding stale flush requests ensures that most flush requests written to SSDs are needed to be flushed regardless of the length of the I/O queues.

The minimal number of parallel flush requests required to hide the speed disparity in an SSD array depends on the hardware configuration of the SSD array. Instead of measuring and setting the minimal number for each SSD array configuration, we only require users to loosely set a maximal number of pending flush requests for an SSD array to avoid having too many flush requests in the queue. We empirically set the maximal number of pending flush requests to $2048\times$ the number of SSDs.

3.3.1 Policy of selecting dirty pages for flushing

The dirty page flusher executes a policy of selecting dirty pages inside each page set. The policy iterates all pages in a page set and assigns a flush score to each page. Thanks to the small size of a page set, there is only small overhead in iterating all pages and computing scores. The current implementation computes scores based on a page eviction policy, given the fact that a dirty page that is more likely to be evicted is more urgent to be flushed to SSDs. The pages that are more likely to be evicted get higher flush scores.

We compute the flush score for GClock [10], one of the page eviction policies supported by SAFS, as follows. We first compute a distance score for each page based on the number of hits and the distance to the clock head.

$$distance\_score = hits \times set\_size + distance$$

We sort the pages based on the distance scores and use the rank of a page in the sorted array as a flush score. The pages with lower distance scores get higher flush scores.

3.3.2 Policy of discarding flush requests

An I/O thread discards flush requests with the following policies: (i) the page in the flush request has been evicted; (ii) the page in the flush request has been cleaned; (iii) the page in the flush request has a flush score lower than a threshold. Discarding flush requests with low flush scores avoids the pages that are likely to be accessed in the future from being evicted by the clean-first page eviction policy.

3.4 Discussion

The flushing scheme maximizes the write throughput but potentially reorders write requests. Therefore, it benefits the applications that allow write reordering. For applications that have more restrict write ordering, we need to introduce a write barrier to SAFS to ensure all writes before the barrier have been written to SSDs. Issuing write barriers frequently diminishes the benefit of the flushing scheme. The applications that require very strict write ordering can hardly benefit from the flushing scheme.

4 Evaluation

We evaluate our design on a non-uniform memory architecture machine with four Intel Xeon E5-4620 processors, clocked at 2.2GHz, and 512GB memory of DDR3-1333. Each processor has eight cores and have hyperthreads enabled. The machine has three LSI HBA controllers connected to a SuperMicro storage chassis, where 18 OCZ Vertex 4 SSDs are installed. Each SSD has 128GB. The machine runs Ubuntu Server 12.04 and Linux kernel v3.2.30.

Due to the complex internal structure and firmware, an SSD may show different performance in different runs even under the same workload. To stablize the I/O performance of an SSD, we write a large amount of data sequentially to the SSD and keep it idle for 10 minutes before each experiment. All I/O throughput is measured when garbage collection on SSDs becomes active.

4.1 Impact of garbage collection

We first explore the impact of garbage collection on an SSD and an SSD array. We conduct experiments with random workloads to explore the following questions:

- Question 1: how does disk occupancy of an SSD affect garbage collection?
- Question 2: how does the number of SSDs in an array affect the throughput of the array when garbage collection becomes active?

Occupancy	maximal	40%	60%	80%
IOPS	60928	42240	38656	32512

Table 1: The I/O throughput of 4KB random write to an SSD with different disk occupancy under active garbage collection. The maximal throughput is measured when there is no garbage collection.

The number of SSDs	1	6	12	18
IOPS per SSD	38656	37888	33280	31744

Table 2: The average I/O throughput of 4KB random write per SSD in arrays of different sizes when each SSD is under active garbage collection. The number of parallel writes per SSD is 128.

- Question 3: what is the minimal number of parallel writes required to achieve the maximal throughput of an SSD array under active garbage collection?

For question 1, we conduct experiments that write 60GB with 4KB uniformly random writes to an SSD and show the result in Table 1. We measure I/O throughput when the SSD is 40%, 60%, 80% full. Table 1 shows that when garbage collection becomes active, the SSD filled with more data has lower I/O throughput. It means that garbage collection becomes more active when an SSD is filled with more data. Garbage collection affects write throughput in all tests.

For question 2, we conduct experiments that dump data to 6, 12, 18 SSDs, attached to 1, 2, 3 HBAs, respectively, and the result is shown in Table 2. Each experiment writes 40GB to each SSD with 4KB random writes. All SSDs are 60% full, and each SSD allows to have 128 pending I/O requests. Table 2 shows that the I/O throughput of each individual SSD decreases as the number of SSDs in the array increases. The result is expected. When more SSDs are installed in the array, more SSDs can interfere the performance of the array. We expect the performance of the array will further decrease when more SSDs are installed.

For question 3, we conduct experiments that write data to 18 SSDs under uniformly random and the Zipfian write-only workloads and vary the number of parallel writes. Figure 2 shows that the I/O throughput increases by up to 28% when the number of parallel writes increases. With a sufficiently large number of parallel writes, we can eventually reach the same performance as each SSD being accessed independently. I/O access patterns can affect the number of parallel writes required to achieve good throughput. Zipfian random workloads require 2304 parallel writes in the SSD array to reach approximately 95% of maximal throughput. In contrast, uniformly random workloads need 9216 parallel writes or even more. Nevertheless, we need to use thousands of or tens of thousands of parallel writes to hide the speed disparity of individual SSDs caused by garbage collection. Based on this experiment and the previous one, we expect that the number of parallel writes required to achieve good performance increases super-linearly with the number of SSDs in an array.

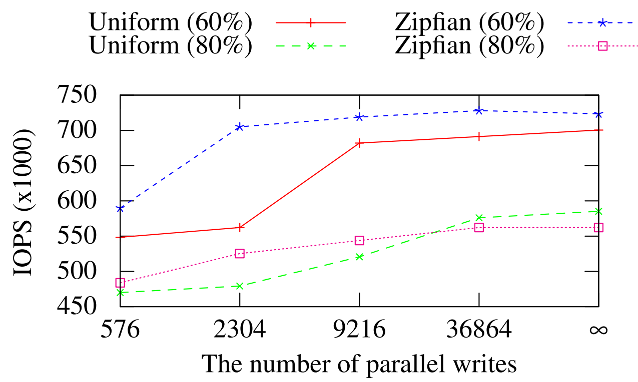

Figure 2: The I/O throughput of 4KB random write to an array of 18 SSDs with different numbers of parallel writes under uniformly random and Zipfian random workloads.

4.2 Effectiveness of the dirty page flusher

We measure the effectiveness of the dirty page flusher by benchmarking SAFS under uniformly random write workloads and Zipfian random write workloads with and without the dirty page flusher enabled. We measure the I/O throughput improved by the dirty page flusher, as well as the amount of extra data writeback caused by the flusher and the cache hit rate. We evaluate both synchronous and asynchronous I/O. Asynchronous I/O uses I/O depth of 32 per SSD. All SSDs are 80% full.

We measure the I/O throughput of asynchronous writes and synchronous writes under write-only random workloads. Figure 3 shows the I/O throughput of aligned random writes. When the dirty page flusher is enabled,

Read percentage	80%	60%	40%	20%	0%
Extra writeback	2.4%	1.6%	2.2%	2.7%	3.2%
Cache hit increase	0.7%	0.6%	1%	1.4%	4%

Table 3: The amount of extra dirty data writeback and the improvement of cache hit rate by the dirty page flusher under Zipfian random workloads with different read/write ratios, compared with cached I/O without the dirty page flusher. Each read/write is 4KB.

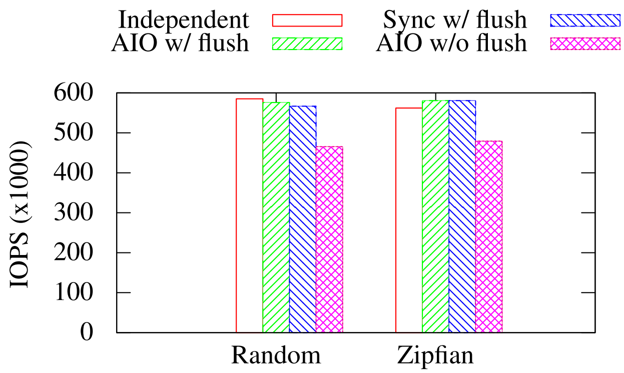

Figure 3: The I/O throughput of SAFS synchronous and asynchronous 4KB random write with and without the dirty page flusher under the uniformly random and Zipfian random workloads. We also include the throughput that all SSDs are written independently.

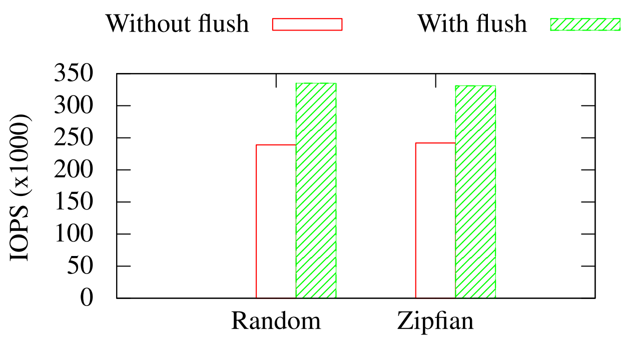

Figure 4: The I/O throughput of SAFS asynchronous write under uniformly random and Zipfian random workloads of unaligned writes. Each write is 128 bytes. We compare the throughput with and without the dirty page flusher.

both synchronous and asynchronous writes can achieve maximal performance (when data is written to SSDs independently), and improve the I/O throughput by up to 24% than that without the dirty page flusher. Figure 4 shows the I/O throughput of unaligned random write. Each write triggers a page read from the SSD array, so synchronous I/O cannot achieve good performance and is not shown in Figure 4. The dirty page flusher can improve I/O throughput of asynchronous write by up to 39%.

We measure the I/O throughput of asynchronous I/O under the uniformly random workloads with different read/write ratios (Figure 5). The dirty page flusher effectively cleans up dirty pages and writes them back to faster SSDs when some SSDs are slowed down by garbage collection. When garbage collection ceases, the page cache

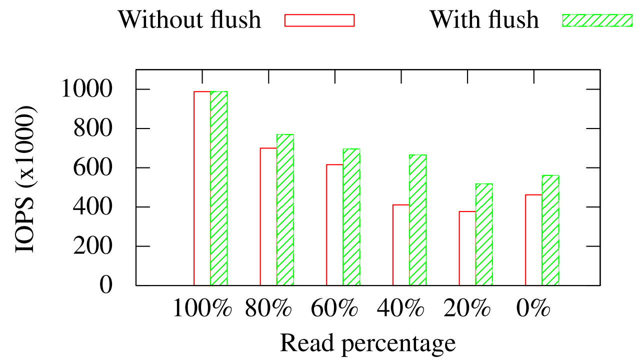

Figure 5: The I/O throughput of SAFS asynchronous I/O under the uniformly random workloads with different read/write ratios. Each read/write is 4KB.

absorbs writes and give reads more opportunity to be issued to SSDs. The flusher improves I/O throughput even when read percentage is high. The largest improvement occurs at read percentage of 40%. The read/write throughput is improved by 62%.

We measure the amount of extra data written back and the cache hit rate affected by the dirty page flusher under Zipfian random workloads with different read/write ratios (Table 3). We compare its result with cached I/O without the dirty page flusher. Although the flusher can cause extra data written back, the amount of extra write-back is fairly small. Furthermore, the flushing scheme slightly increases the cache hit rate because it helps evict dirty pages that are unlikely to be accessed again.

5 Conclusions

We propose a software solution that tackles unsynchronized garbage collection in an SSD array. We maintain long I/O queues in the main memory for each SSD and use a dirty page flusher to pre-clean dirty pages and fill the long I/O queues. We define a policy of selecting dirty pages to flush and a policy of discarding stale flush requests to reduce the amount of data flushed to SSDs.

We evaluate the design with uniformly random and Zipfian random workloads. The design improves the I/O throughput by up to 28% under write-only workloads, and by up to 62% under uniformly random mixed read/write workloads. We further demonstrate that the design causes little extra data written back to SSDs and slightly improves the cache hit rate under Zipfian random workloads.

References

[1] Gupta, A., Kim, Y., and Urgaonkar, B. DFTL: a flash translation layer employing demand-based selective caching of

page-level address mappings. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems* (New York, NY, USA, 2009).

[2] JUNG, H., SHIM, H., PARK, S., KANG, S., AND CHA, J. LRU-WSR: integration of LRU and writes sequence reordering for flash memory. *IEEE Trans. on Consum. Electron. 54*, 3 (2008).

[3] KIM, H., AND AHN, S. BPLRU: a buffer management scheme for improving random writes in flash storage. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2008).

[4] KIM, Y., ORAL, S., SHIPMAN, G. M., LEE, J., DILLOW, D. A., AND WANG, F. Harmonia: A globally coordinated garbage collector for arrays of solid-state drives. In *Proceedings of the 2011 IEEE 27th Symposium on Mass Storage Systems and Technologies* (Washington, DC, USA, 2011).

[5] LEE, S., SHIN, D., KIM, Y.-J., AND KIM, J. LAST: locality-aware sector translation for nand flash memory-based storage systems. *SIGOPS Oper. Syst. Rev.* (2008).

[6] LEE, S.-W., AND MOON, B. Design of flash-based DBMS: an in-page logging approach. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 2007).

[7] LV, Y., CUI, B., HE, B., AND CHEN, X. Operation-aware buffer management in flash-based systems. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data* (New York, NY, USA, 2011).

[8] MIN, C., KIM, K., CHO, H., LEE, S.-W., AND EOM, Y. I. SFS: random write considered harmful in solid state drives. In *Proceedings of the 10th USENIX conference on File and Storage Technologies* (Berkeley, CA, USA, 2012).

[9] PARK, S.-Y., JUNG, D., KANG, J.-U., KIM, J.-S., AND LEE, J. CFLRU: a replacement algorithm for flash memory. In *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems* (New York, NY, USA, 2006).

[10] SMITH, A. J. Sequentiality and prefetching in database systems. *ACM Trans. Database Syst. 3*, 3 (1978).

[11] ZHENG, D., BURNS, R., AND SZALAY, A. S. A parallel page cache: IOPS and caching for multicore systems. In *USENIX conference on Hot Topics in Storage and File Systems* (2012).

[12] ZHENG, D., BURNS, R., AND SZALAY, A. S. Toward millions of file system IOPS on low-cost commodity hardware. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Denver, Co, USA, 2013).